Palo Alto Research Center

A Necessary and Sufficient Condition for the Existence of Hoare Logics

By Richard J. Lipton

XEROX

A Necessary and Sufficient Condition for the Existence of Hoare Logics

by Richard J. Lipton

CSL-77-4 JUNE 1977

XEROX
PALO ALTO RESEARCH CENTER
3333 Coyote Hill Rd / Palo Alto / CA 94304

Abstract

A necessary and sufficient condition is obtained for the existence of Hoare Logics for a large class of programming languages. In addition, Cook's concept of expressiveness is shown to be a very powerful restriction on the assertion language.

Key Words and Phrases:

complete and sound, decidable halting problem, expressive assertion language, Hoare logic, programming language, schemata

CR Categories

5.2

1. Introduction

1. Introduction

The important paper of Cook[3] on the existence of a relative complete and sound Hoare logic for a fragment of Algol has led many researchers to consider the existence of such logics for other programming languages[2,5,7,8,9,10,11]. In particular, Clarke[2] and Lipton-Snyder[9] have shown that a programming language **P** can have no relative complete and sound Hoare logic unless the programs of **P** have a decidable halting problem over finite interpretations. (This result was first observed implictly in [2].) Here our main result is the extension of this result to

Theorem 1: *For any acceptable programming language* **P** *the following are equivalent*:

(1) **P** *has a decidable halting problem for finite interpretations.*

(2) **P** *has a relative complete and sound Hoare logic.*

A definition of these concepts -especially *acceptable*- is contained in the next section. We note now, however, that acceptable is a mild restriction that is satisfied by a wide class of programming languages.

The main importance of this theorem is that it links familiar ideas (condition 1) with new and perhaps less familiar ideas (condition 2). Indeed, at this point it is perhaps worth digressing and discussing in some detail just what a relative complete and sound Hoare logic is.

Let **P** be a programming language in the sense that **P** is a collection of schemata (this is made precise in the next section.) For example, **P** might be the class of flowchart schemata or the class of call-by-name recursive schemata. Then an interpretation **I** assigns meaning to the uninterpreted predicates and functions of *P*. Now we use

$$\models_{\mathbf{I}} A\{P\}B$$

to mean that if program *P* is started in some initial state that satisfies the predicate *A*, then either *P* diverges or else it halts in some state that satisfies *B* - all this with respect to the fixed interpretation **I**. Moreover, as usual it is assumed that both *A* and *B* lie in a first order theory - the so called *assertion language* of P. Then a Hoare logic is defined to be a

finite collection of axioms and rules of inference; let

$$\vdash_{\mathbf{H}} A\{P\}B$$

mean that we can prove $A\{P\}B$ in this logic. Then the notion of sound should be clear,

$$\vdash_{\mathbf{H}} A\{P\}B \quad \text{implies} \quad \models_{\mathbf{I}} A\{P\}B.$$

Cook's insight was to attempt to obtain the converse to soundness, i.e. completeness. Unfortunately, it is impossible to obtain this and retain the standard restriction that the set of theorems of the Hoare logic be recursively enumerable. This follows immediately from the fact that the set of divergent programs (programs such that $\models_{\mathbf{I}} T\{P\}F$ where T and F are the logical constants) is not always recursively enumerable. Cook then proceeded to expand the notion of a Hoare logic. He now allowed the logics the ability to "ask" an oracle about the truth of an arbitrary first order sentence from the assertion language. Then he was able to show that

$$(*)\ \models_{\mathbf{I}} A\{P\}B \quad \text{iff} \quad \vdash_{\mathbf{H,I}} A\{P\}B.$$

Well almost. Actually, one further and quite important assumption was needed: the so called notion of *expressibility*. Roughly, the assertion language is expressive for the interpretation I provided for any program P the set of initial states, say A, that cause P to diverge is first order definable (in the assertion language.) One very important point is that we do not require that there be an effective way to construct A from P just that A be first order definable. Then, provided that the assertion language was expressive, Cook was able to prove (*).

With these general comments out of the way, let us return to the discussion of theorem 1. As we stated before, the main importance of this result is the implication that allows us to assert the existence of a relative complete and sound Hoare logic. Until now, the quest for such logics was always *ad hoc*: given a new programming language one just attempted to find, in someway, the right axioms and rules of inference.

Theorem 1 is also interesting since its proof rests on a fact about expressiveness that is of independent interest. In particular, it rests on a generalization of an earlier result of DeMillo-Lipton-Snyder [5]:

Theorem 2: *If the assertion language, say* L_A, *is expressive for the interpretation* I *then either*

(1) *a standard model of arithmetic can be defined in* L_A *or*

(2) $\forall P \in \mathbf{P} \exists n$ *such that P reaches at most n states in any computation (divergent or not) with any initial state.*

The notion of state as used here requires some further comment: it is defined precisely in the next section. The state of a program (in an acceptable programming language) is determined by the information that can be *immediately* accessed. Thus if P is a recursive program this information consists of all the current values of the variables of P but not the contents of the whole stack of P.

Roughly, theorem 2 states that in order for the assertion language to be expressive either it must contain all of arithmetic or there must be no program (over the given interpretation) that reaches more than some bounded number of states. This theorem is hinted at in Cook's original paper when he points out that Presburger Arithmetic is not expressive. While it is perhaps not surprising that theorem 2 is true, it is interesting to remark that at several places in the literature it is suggested that expressiveness is not too strong an assumption. We simply wish to observe that except for "degenerate" cases (i.e. when condition 2 of theorem 2 is true) it is quite strong.

Finally, we would like to add some comments on the motivation of studying Hoare logics especially since we have in the past attacked program verification (DeMillo-Lipton-Perlis [4]). We have felt then and still believe now that formal verification of computer programs can only fail. This does not mean however that the mathematical structure of computer programs is not of interest. But we must never confuse the ability to doing something in principle with the ability to do it in practice.

2. Basic Definitions

2. Basic Definitions

In this section we will first explain what an acceptable programming language is. Then we will go over the key concepts of Hoare Logic already discussed in the introduction.

We would like to avoid completely the issue of defining the concept of an acceptable programming language. If everywhere we use this term the reader simply substitutes either flowcharts or recursive programs (with any reasonable parameter passing method) then he will not go wrong. All such programming languages are indeed acceptable in our sense. However, in order to be quite precise and also in an attempt to state our theorems in as general a way as possible we will proceed with a definition of the concept of an acceptable programming language. (A word of *caution*: some of the more interesting problems in proving theorems 1 and 2 do not arise in the simple case of flowcharts).

The key to all this is that we define a program P by giving a recursive procedure P^* - which we will call the *interpreter* of P. Essentially P^* supplies the operational semantics of what P is to do for any interpretation **I**. Do not confuse P^* and **I**: **I** assigns meaning as usual to uninterpreted functions and predicates, while P^* assigns a meaning to what P does in a particular interpretation **I** and when started in a particular initial state. We will now detail exactly how P^* defines the computation of the program P once the interpretation **I** is fixed and some initial state is fixed. First by a *state* of P we mean an element of the form

$$\langle s, x_1, \ldots, x_k \rangle$$

where s is in the finite set of *statements* of P and the values of each x_i are always values from the given interpretation **I**. As usual, **I** assigns meaning to the uninterpreted functions and predicates of P. A *computation* over **I** with inputs $x_1, \ldots, x_k$ is a sequence of states $S_0, \ldots, S_m$ such that

1. S_0 is $\langle s_0, x_1, \ldots, x_k \rangle$ where s_0 is the *initial statement* of P.

2. S_{n+1} is obtained by P^* as follows: First P^* is allowed to ask questions of the form $A(w_1, \ldots, w_k)$

where A is a predicate letter and each w_j is a value from earlier states S_i with $i<n+1$. Then P^* computes $S_{n+1} = \langle s,y_1,...,y_k \rangle$ by

a) setting s to some statement of P;

b) setting each x_i to some $f(w_1,...,w_k)$ where f is a function letter and each w_j is as before.

Finally, *convergence* and *divergence* are defined in the obvious way: a state $\langle s,x_1,...,x_k \rangle$ is *final* provided s is the *final statement* of P; in this case the *output* is $x_1,...,x_k$.

Without proof we will now claim that many programming languages are definable in this way. In particular, each of the following can be so defined:

1. flowcharts;

2. recursion with call-by-value parameter passing;

3. recursion with call-by-name parameter passing;

and many others. The first we note is the class studied by Cook [3], the second by Gorelick [6], and the third by Lipton-Snyder [7].

We now are ready to define the notion of an acceptable programming language. We say that **P** is an *acceptable programming language* provided there is a finite collection of uninterpreted functions and predicates such that the following are true:

1. Each program P in **P** is defined by an interpreter over these functions and predicates;

2. All flowharts are in **P**;

3. **P** is closed under *subroutine calls.*

The last concept of subroutine calls is quite awkard to state precisely: intuitively it means that **P** allows the "insertion" of one program into another. Let P and Q be two programs in **P**. For ease of notation we assume that the states of P are of the form $\langle s,x_1,...,x_k \rangle$ with s in $\{s_1,...,s_p\}$ and the states of Q are $\langle t,x_1,...,x_k \rangle$ with t in $\{t_1,...,t_q\}$. Further, we assume that the initial statements are s_1 and t_1 and the final statements are s_p and t_q. Then closure under subroutine calls allows us for any s_m (a statement of P) to effectively find a program R in **P** such that

1. the states of R are $\langle r,x_1,...,x_k\rangle$ where r is in $\{s_1,...,s_p,t_1,...,t_q\}$;

2. the initial (final) statement of R is $s_1(s_p)$;

3. the interpreter R^* of R operates on $S_0,...,S_n = \langle r,x_1,...,x_k\rangle$ as follows

   a) if $r = s_i$ and $i \neq m$ then it essentially simulates P^*;

   b) if $r = s_m$ then it sets r to t_1;

   c) if $r = t_i$ and $i \neq q$ then it essentially simulates Q^*;

   d) if $r = t_q$ then it essentially simulates P^* as if s was equal to s_m.

Roughly, R behaves as if a subroutine call was made from program P to Q just before statement s_m was reached; if Q converges then control resumes with statement s_m. Note that there is really no parameter passing here; P and Q communicate simply by sharing all their variables. In a sense this is not at all a subroutine mechanism but rather a simple type of closure under macro's.

We now observe that flowcharts, recursion with call-by-value, and recursion with call-by-name all are acceptable programming languages.

Let us now fix a particular acceptable programming language **P**. We now will just quickly go over the necessary concepts (see for example Clarke [2] or Cook [3]):

1. L_A is the assertion language which we assume for simplicity consists of exactly the predicates and functions of **P**. As in Cook [3] we assume that equality is in this language.

2. We use **I** to denote an interpretation. However, we further assume that **I** is *effectively presented*. That is, each function and predicate is given by a recursive function or predicate. Clearly, in terms of realistic examples this restriction is certainly reasonable.

3. As already discussed in the introduction we use the standard triple notation $A\{P\}B$; also $\models_{\mathbf{I}} A\{P\}B$ means that this triple is true over the interpretation **I**.

4. The assertion language L_A is *expressive* for the interpretation **I** provided for all programs P there is a first order sentence A in L_A such that $\models_{\mathbf{I}} A\{P\}F$ and moreover if $\models_{\mathbf{I}} B\{P\}F$ then B implies A. This definition may at first glance appear to be weaker than the standard one it can easily be shown to be equivalent when A and B are computable predicates. In any case this restriction is a reasonable one.

5. A Hoare logic H is defined as in Cook [3] (see also section 3); we use $\vdash_{\mathbf{H,I}} A\{P\}B$ to mean that this triple is provable in the Hoare logic. Finally, this logic is *relative sound and complete* provided for all interpretations I, if L_A is expressive then for all programs P,

$$\models_{\mathbf{I}} T\{P\}F \quad \text{iff} \quad \vdash_{\mathbf{H,I}} T\{P\}F$$

As in our definition of expressive this can be easily shown to be equivalent to the usual notion of relative sound and complete even though we restrict our attention to triples of the form $T\{P\}F$ provided A and B are computable.

One final comment. As in schemata theory the *Herbrand Universe* refers to the set of terms over the function letters of P; the *depth* of a term and its *value* with respect to a given interpretation are defined as usual.

3. Main Results

3. Main Results

In this section we will prove our theorems. Therefore let us agree to fix **P** to denote some acceptable programming language. Even though theorem 1 logically depends on theorem 2 it is convenient to first prove it with the assumption that theorem 2 is true.

In order to prove theorem 1 we must show that the following are equivalent:

(1) **P** has a decidable halting problem for finite interpretations.

(2) **P** has a relative complete and sound Hoare logic.

The fact that (2) implies (1) is in Clarke [2] and Lipton-Snyder [7] and is omitted. We therefore turn our attention to demonstrating that (1) implies (2).

At this point it is important to be quite clear just what a Hoare logic is since we are going to prove the existence of one. What we intend to show is that provided (1) is true there is a recursive search procedure that will possibly from time to time output a triple $A\{P\}B$. This procedure will be able to ask an oracle arbitrary an first order sentences of L_A about the interpretation **I**. The key constraints on this procedure are:

soundness: if $A\{P\}B$ is output, then $\models_{\mathbf{I}} A\{P\}B$.

completeness: if $\models_{\mathbf{I}} A\{P\}B$, then it is eventually output.

We further assume that our search procedure has access to the interpretation **I** in that it is given the effective presentation of **I**. In addition, as we did in our definitions of expressive and relatively sound and complete we will further restrict ourselves to triples of the form $T\{P\}F$.

Our search procedure is most easily thought of as divided into two subprocedures. Each of these subprocedures will be sound; and together they will be complete. Moreover, it is also convenient to assume that each of these procedures operates by dovetailing on each triple $T\{P\}F$. Thus for the remainder of this proof we need only concern ourselves with what these procedures do on this one fixed triple.

Before we can study these procedures we need some simple notation. Let $P\langle x\rangle$ denote the set of states that P visits in the computation with $x = x_1,...,x_k$ as input. Also let $|P\langle x\rangle|$ denote the cardinality of $P\langle x\rangle$. (Recall that we have fixed the interpretation **I**.)

The first procedure essentially guesses that $|P\langle x\rangle|$ is always less than say M for any x. Now we claim that there is an effectively constructible sentence Δ such that Δ is true in the interpretation **I** if and only if P diverges everywhere *and* our assumption about $|P\langle x\rangle|$ was correct. Then, if Δ is true this procedure will output the triple $T\{P\}F$. Thus whether this guess is correct or not this procedure is always sound. It remains to argue why such a sentence must exist. Let $U_M(x)$ denote the set of values obtained from $x = x_1,...,x_k$ by terms of depth at most M. Then the key observation is the following: if $\langle s,y_1,...,y_k\rangle$ is a state in $P\langle x\rangle$ then each y_i lies in the set $U_M(x)$. This is based on the following simple lemma:

Lemma: *Let* $t = t_0,t_1,...$ *be a sequence of terms from the Herbrand Universe of* **P** *with respect to the variables* $x_1,...,x_k$. *Assume also that each* t_i *is either one of these variables or is equal to* $f(w_1,...,w_l)$ *where each* w_j *is a member of the sequence* t *that occurs before* t_i. *Further assume that with respect to the interpretation* **I** *there are exactly* M *values in the sequence* t. *Then every* t_i *has a value that can be obtained by some term of depth at most* M.

Proof of Lemma: We will prove this by induction on M. If M is equal to 1 then clearly this value must be equal to t_0 and this is a variable and hence is of depth at most M. Now let $M>1$. Let t_i be the first time that the Mth value occurs. Then either it is a variable and we are done or it is equal to $f(w_1,...,w_k)$ where each w_j occurs earlier. But then by induction each w_j is of depth at most M-1; hence, t_i is of depth at most M.□

Let y_i be as above. Then we can apply the lemma to the sequence t which is the sequence of values of the variables of the computation of P. Then it follows that y_i lies in $U_M(x)$. Now let $\mathbf{I}_1,\mathbf{I}_2,...$ be all possible interpretations with universes $\mathbf{U}_1,\mathbf{U}_2,...$ of size less than or equal to N where N is 1+the size of the Herbrand Universe of depth M. We assume that each universe has one distinguished element which we denote by *. We will say that the interpretation **I** is *isomorphic* on $U_M(x)$ to $\mathbf{I}_i$ provided there is a mapping φ from $U_M(x)$ to $\mathbf{U}_i$-{*} such that

a) $A_{\mathbf{I}}(w_1,...,w_l)$ iff $A_{\mathbf{I}_i}(\varphi w_1,...,\varphi w_l)$ for all $w_1,...,w_l$ in $U_M(x)$ (subscripts denote that the value is with respect to the given interpretation.);

b) if $f_{\mathbf{I}}(w_1,...,w_l)$ is not in $U_M(x)$ for $w_1,...,w_l$ in $U_M(x)$ then $f_{\mathbf{I}_i}(\varphi w_1,...,\varphi w_l)$ is equal to *.

Note that * represents all the universe of **I** that is not in $U_M(x)$. Now let $\Delta_i(x)$ be "**I** is

isomorphic on $U_M(x)$ to $\mathbf{I}_i$", and let Δ be"$\forall x \vee \Delta_i(x)$"where the disjunction is over all those $\mathbf{I}_i$ such that for any input $\langle x_1,...,x_k \rangle$ where each x_j is in $\mathbf{U}_i$-{*}, P diverges and moreover no state in any of these computations contains the value *. Then it is not hard to see that Δ is the required sentence. In order to see this first let $\Delta_i(x)$ be true. Then P diverges on x. This follows since $\mathbf{I}$ is isomorphic on $U_M(x)$ to $\mathbf{I}_i$ and P diverges on $\mathbf{I}_i$ without ever using *; hence, P's computation on $\mathbf{I}$ must be the same and it must diverge. On the other hand, let P diverge. Then it does so by only using values from $U_M(x)$, and hence P must diverge on some $\mathbf{I}_i$ without ever using the value *.

We will now describe the second search procedure. It will guess that there is no bound M on $|P\langle x\rangle|$ for all inputs x. Then by theorem 2 we can conclude that arithmetic *is* definable in L_A. The idea now is quite simply: we plan to encode the truth of the triple $T\{P\}F$ into a first order sentence of arithmetic. The problem with this plan is that we only assert that arithmetic is definable there need be *no effective* way to find just how it is defined in L_A. Therefore the way this second procedure will operate is as follows:

1. first it will guess formulas for the notions of arithmetic needed;

2. then it will check that these formulas satisfy a suitable finite set of axioms;

3. finally if the axioms are satisfied then the procedure will encoded the truth of the triple into arithmetic and then ask its oracle.

There are still some problems with this outline. The main difficulty is that no finite set of axioms can characterize all of arithmetic; hence, *nonstandard models* will come into play. We will in a moment show by careful analysis how one can overcome this difficulty.

As stated above this procedure will first guess formulas for the necessary concepts from arithmetic. These are:

"x is a *natural number*", "x *equals* y", "y is the *successor* of x", "$x<y$", "$x+y=z$", and "$x \times y=z$".

There is one further complication, as we will see in a moment theorem 2 does *not* provide that natural numbers will be represented by simple variables of the assertion language. Instead natural numbers are represented by *pairs* of such variables. We therefore will make the following convention for the rest of this section: **x**, **y**, and **z** will stand for respectively (x_1,x_2), (y_1,y_2), and (z_1,z_2). Then will we *guess* formulas:

$N(\mathbf{x})$, $E(\mathbf{x},\mathbf{y})$, $S(\mathbf{x},\mathbf{y})$, $L(\mathbf{x},\mathbf{y})$, $A(\mathbf{x},\mathbf{y},\mathbf{z})$, and $M(\mathbf{x},\mathbf{y},\mathbf{z})$

(repectively, "x is a *natural number*", "x *equals* y", "y is the *successor* of x", "x<y", "x+y=z", and "x×y=z".)

Next we will consider the following set of axioms N_1:

1. $x' \neq 0$
2. $x'=y' \Rightarrow x=y$
3. $x+0=x$
4. $x+y'=(x+y)'$
5. $x \times 0=0$
6. $x \times y'=(x \times y)'+x$
7. $\neg(x<0)$
8. $x<y \leftrightarrow x<y \vee x=y$
9. $x<y \vee x=y \vee y<x$.

By Shoenfield [12] sections 4.6, 4.7, and 6.7, we can find a set of formulas N_2 over N, E, S, L, A, and M such that for any recursive set F we can effectively construct a formula again over these formulas such that

$$i \in F \text{ implies } \vdash_{N_2} K_i(\mathbf{x}) \rightarrow \alpha(\mathbf{x})$$

and

$$i \notin F \text{ implies } \vdash_{N_2} K_i(\mathbf{x}) \rightarrow \neg\alpha(\mathbf{x})$$

where $K_i(\mathbf{x})$ if and only if x is equal to the ith sucessor of 0. Note, this set of axioms is different from N_1 simply because we have predicates and *not* functions for the notions such as successor. Now we will ask the oracle whether or not the formulas N_2 are true in the interpretation I. If they are then we will ask the oracle whether or not

$$(\dagger) \quad \forall \mathbf{x}\{N(\mathbf{x}) \rightarrow \alpha(\mathbf{x})\}$$

is true in I where α encodes the set $F=\{(i,j) \mid P$ does not converge in $< i$ steps on the j input$\}$. Here (i,j) denotes some pairing function. (Note: this is the only place where we use that I is effectively presented.) If both these are true then this procedure will output the triple $T\{P\}F$.

It remains to show that this second procedure is always sound and that both procedures together are complete. We will first show that it is sound. Thus let us assume that P converges on some input j in i steps. Therefore, by the definition of the set F,

$$\vdash_{N_2} K_k(x) \rightarrow \neg\alpha(x)$$

where $k=(i,j)$ and so

$$\models_I K_k(x) \rightarrow \neg\alpha(x).$$

But in the interpretation I by (†), $N(x) \rightarrow \alpha(x)$, and this is a contradiction since it also follows that $K_k(x) \rightarrow N(x)$ and $K_k(x_0)$ for some x_0 are true; hence, both $\alpha(x_0)$ and $\neg\alpha(x_0)$ are true. Thus the second search procedure is sound. Note, the keys to this argument are that on the *standard numbers* the predicate $\alpha(x)$ is well behaved and that the question we need to ask our oracle about (†) is a *universal sentence.*

In order to complete the proof of our results we need only prove theorem 2 for this will show that the two search procedures together are complete. Recall that we must show that if there is a program say P such that $|P\langle x\rangle|$ is not bounded, then arithmetic can be defined in L_A. Therefore we will assume that there is such a program. In order to understand the proof of this theorem it is convenient to at first make a much stronger assumption:

> (*) there is an input x_0 such that the computation with this input reaches an infinite number of *distinct* states.

We will now, as in DeMillo-Lipton-Snyder [5], show that arithmetic is definable in L_A. Clearly, there is some variable say x_i that in this computation it takes on an infinite number of distinct values. One is then tempted to use this variable as a "sucessor function": in order to find the sucessor of z just run P until x_i equals z and then the next new value of x_i is the sucessor of z. The difficulty with this idea is that x_i may take on the same values many times. (It may be interesting to note at this point why flowcharts are much simpler than an arbitrary acceptable programmming language. If P was a flowchart then one could use the entire state of P as the value of the counter: P cannot repeat this state without entering a loop. Clearly, in general this is not the case; hence, the extra complexity when P is in an arbitrary acceptable programming language.) In order to overcome this difficulty let us look at our problem (x_i repeating values) in a quite general way. Assume that we are given a "bad stop watch" C with the following properties:

1. C has a *display window* which allows us to see what "time" it is. (We can assume that in this window is an element from some arbitrary universe; we can only test equality on this universe; in particular there is no ordering.)

2. C has a *reset button* which allows us to start the watch over at the beginning.

3. C has a *step button* which allows us to advance it to the next time. The key is that we do not assume that the time displayed is always new. We just assume that the times displayed run through an unbounded number of values.

Then our task is to construct a good stop watch C' with all the properties of C except that it never repeats a time. Note it is definitely unfair to remember all the pass history of the watch. We are only allowed to remember a bounded number of past times.

The good stop watch is constructed out of two identical copies say C_1 and C_2 of the bad stop watch C. On reset C' displays the same time as C. Now suppose that C' is currently displaying t; we will desribe how to find the next time that it will display:

1. Run the first stop watch C_1 until it displays t.

2. Now run this watch until it displays a time different from t and let this value be t'.

3. Now start the other stop watch C_2 until either it displays t or t': (do *not* reset C_1.)

   a) If it first displays t then the next time of C is t'.

   b) If it first displays t' then we return to step 2.

We now claim that C' never repeats a time; actually we claim more: C' goes through each time that C goes through but C' simply has eliminated all the repeats. This is easily seen by induction. Moreover, we can use this stop watch construction to solve our original problem. Now the role of the bad stop watch is played by the program P with x_i as the value it displays. The crucial point then is that the program P' that corresponds to the good stop watch is in **P** since **P** is an acceptable programming language. This follows since the definition of C' only uses a simple subroutine call and so can be handled by any acceptable language. Note: if in step 3 of the construction of C' we did not always start C_2 off at the beginning then we would not have necessarily had that P' was in **P**.

All the above discussion has given us a program P' that acts like a successor function: on input x it steps through the sucessors of x. We will now show how to use it to define arithmetic in L_A. Remember however that all of this is with respect to the strong assumption (*); in a moment we will show how to weaken this assumption. We will now combine the expressiveness of L_A with the program P' to obtain the existence of all the required formulas:

1. We obtain $N(x)$ as follows. Consider the program that on input x searches to see if x equals $n_0,n_1,...$ (where n_0 is x_0 and n_i is the next output of P' given n_{i-1} as input.) Then $\neg N(x)$ if and only if this program diverges, and so by expressiveness $N(x)$ is first order definable.

2. We obtain the other formulas by constructing similiar programs and then appealing to expressiveness.

The last step necessary to complete the proof of theorem 2 is to show that we do not need the strong assumption (*) but instead need only that $|P<x>|$ is unbounded. If we now only assume this then we can proceed as before except that now numbers are encoded as pairs (x_1,x_2). We construct by our "stop watch argument" a new program P' as before. We now use (x_1,x_2) to denote the number k if and only if

P' started on input x_1 reaches output x_2 in exactly k steps.

Note this construction of numbers is just like the usual construction of the rationals as pairs of integers. Then just as before we can appeal to the existence of P' and the expressiveness of the assertion language to obtain the existence of all the neeeded formulas. We will, therefore, only sketch one part of this argument: the existence of $E(\mathbf{x},\mathbf{y})$. We construct a program that operates on inputs $\mathbf{x}=(x_1,x_2)$ and $\mathbf{y}=(y_1,y_2)$ as follows.

1. $z \leftarrow x_1$ and $w \leftarrow y_1$.

2. If $z=x_2$ and $w=y_2$ then halt.

3. Let $z \leftarrow$the result of running P' on input x_1 until it reaches the first step after z; let $w \leftarrow$the result of running P' on input y_1 until it reaches the first step after w. Then goto step 2.

Then this program diverges if and only if $\neg E(\mathbf{x},\mathbf{y})$, and so as before E is first order definable. Note that this and all other such programs are in $\mathbf{P}$ because $\mathbf{P}$ is an acceptable programming language. This is why we could not in the above construction just alternatively run two copies of P': such a construction is not always doable in an acceptable programming language.

This completes the proofs of both our theorems.

4. Conclusions

4. Conclusions

We would like to make several remarks about our main theorems. First, theorem 1 can be used to yield the results of Cook [3] for flowcharts, Gorelick [6] for recursion with call-by-value recursion, Clarke [2] for co-routines, and Lipton-Snyder [7] for recursion with call-by-name. Of course just matching known results is not that important; however, theorem 1 can be used to obtain new results. For example, let us consider the class of *nested stack automata* [1]: roughly these correspond to machines with the ability to enter their stack in read-only mode. Then just as recursion with call-by-value parameter passing corrresponds to push down automata, a new and more powerful class of recursive algorithms corresponds to nested stack automata, let us denote it by S. Now theorem 1 applies and demonstrates that S has a complete and sound Hoare Logic. This is interesting since any direct proof of this fact must reproduce a good deal of the standard and nontrivial proof that nested stack automata have a decidable halting problem.

Our second set of comments concerns itself with a number of points about theorem 2. The literature on Hoare Logics has at times hinted that the notion of expressive was not too powerful: theorem 2 shows that this is not so. Expressiveness is shown by this theorem to be a very powerful restriction on the assertion language. It is almost equivalent to always having the assertion language contain all of arithmetic. This result, therefore, justifies the concentration that is placed on arithmetic as an assertion language in such interesting papers as [7]. Finally, it is easy to see that theorem 2 is best possible in the sense that one cannot hope to prove that condition 2 can be improved to say there is a uniform bound on $|P\langle x\rangle|$ independent of P: this follows from the fact that larger and larger programs can always explore more and more of the Herbrand Universe. Also one can show that the construction used in theorem 2 is necessary for there are interpretations I such that arithmetic is definable and yet no program reaches an infinite number of states in any single computation.

Finally we would like to make several disclaimers about theorem 1. We do indeed show in a precise sense that Hoare Logics exist for a large class of programming languages. However, this result must be evaluated carefully. First, we are forced to make several technical restrictions. These are: (i) the assertion language must be equal to what is usually called the expression language, (ii) the programming language must be acceptable, and (iii) the notion of Hoare logic is that of a recursive search procedure. The first two of these are

not at all unreasonable. Perhaps the one that we would most like to relax is the part of the definition of an acceptable programming language that requires that the programs be deterministic. Clearly, this is critical to our proof; otherwise, all our "stop watch" constructions would fail (one could however still obtain our results in the case that the programs were nondeterministic but determinate.) However, the most important restriction of all is certainly the notion of Hoare logic that is used here. The assumption that the notion of a logic can be replaced by a search procedure is of course not new; however, other notions of just what a Hoare logic is are possible. We still believe that results such as those obtained here will help us understand the basic structure of program logics. In particular, we hope that the restriction that the interpretation must be effectively presented can be dropped at least in the presence of some other mild restriction.

Acknowledgements: I gratefully thank R. DeMillo and L. Snyder for many long and interesting discussions about Hoare logics. I also thank L. Guibas and A.R. Meyer for a number of important observations. Finally, I wish also to thank C. Yap for comments on an earlier version of this paper.

References

[1] A. Aho.
Nested Stack Automata
JACM 1969 vol. 16, no. 3 p383-406.

[2] E.M. Clarke.
Programming language constructs for which it is impossible to obtain "good" Hoare-like axiom systems. *Fourth ACM Symposium of Programming Languages.*

[3] S.A. Cook
Axiomatic and interpretative semantics of an algol fragment. Technical report 79, Computer Science Department, University of Toronto.

[4] R.A. DeMillo, R.J. Lipton, A.J. Perlis
Social Processes in Proving Theorems and Programs, *Fourth ACM Symposium of Programming Languages.*

[5] R.A. DeMillo, R.J. Lipton, and L. Snyder.
In preparation.

[6] G. Gorelick.
A complete axiomatic system for proving assertions about recursive and non-recursive programs. Technical report 75, Computer Science Department, University of Toronto.

[7] D. Harel, A.R. Meyer, and V. Pratt.
Computability and completeness in logics of programs. *Ninth Symposium on Theory of Computing*, pp. 261-268.

[8] D. Harel, A. Pneuli, and J. Stavi.
A complete axiom system for proving assertions about recursive programs. *Ninth Symposium on Theory of Computing*, pp. 249-260.

[9] R.J. Lipton and L. Snyder.
In preparation.

[10] S. Owicki.
A consistent and complete deductive system for the verification of parallel processes. *Eight Symposium on Theory of Computing*, pp. 249-260.

[11] V. Pratt.
Semantic considerations in Floyd-Hoare logic. *17th Foundations of Computer Science*, pp. 109-121.

[12] J.R. Shoenfield.
Mathematical Logic, Addison-Wesley, 1967.

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

XEROX® is a trademark of XEROX CORPORATION Printed in U.S.A.